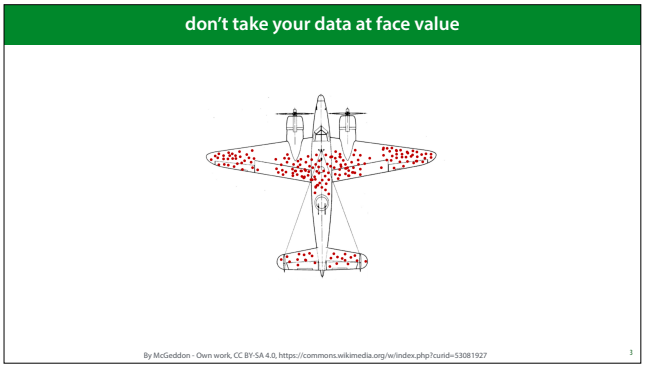
Data pre-processing
Part 1: Missing values and outliers

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

To motivate today's lecture, let's look at a famous historical case of operations research. In the second World War, the allies executed many bombing runs, and often, their planes came back looking like this.

To investigate where they should reinforce their planes, investigators made a tally of the most common points on the plane they were seeing damage.



The initial instinct was to reinforce those places that registered the most hits.

However, it was soon pointed out (by a man called Abraham Wald) that this ignores a crucial aspect of the *source* of the data. They weren't tallying where planes were most likely to be hit, they were tallying where planes were most likely to be hit *and come back*.

The places where they weren't seeing *any* hits were exactly the places that should be reinforced, since the planes that were hit there didn't make it back.

This specific effect is called survivorship bias, and it's worth keeping in mind, but the more general lesson for today, is that you should **not take your data at face value**.

Don't just load your data into an ML model and check the predictive performance: consider what you're ultimately trying to achieve, and consider how the source of your data will affect that goal.

look at your data

	mean of x	mean of y	var. of x	var. of y	correlation between x and y	regression line w	regression line b	r^2
dataset 1	9	7.5	11	4.125	0.816	0.5	3	0.67
dataset 2	9	7.5	11	4.125	0.816	0.5	3	0.67
dataset 3	9	7.5	11	4.125	0.816	0.5	3	0.67
dataset 4	9	7.5	11	4.125	0.816	0.,5	3	0.67

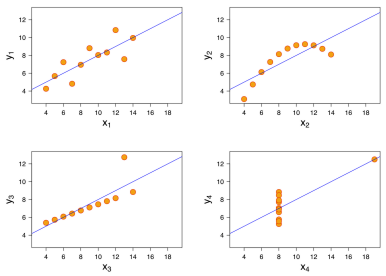
4

Imagine that I gave you four datasets, each with two features x and y. For all datasets all of the following statistics give the same value: the mean and variance of x, the mean and variance of y, the correlation between x and y, the parameters of the linear regression line that best fits, and the r^2 of the correlation.

You would conclude, that the datasets must be pretty similar, right?

---

look at your data



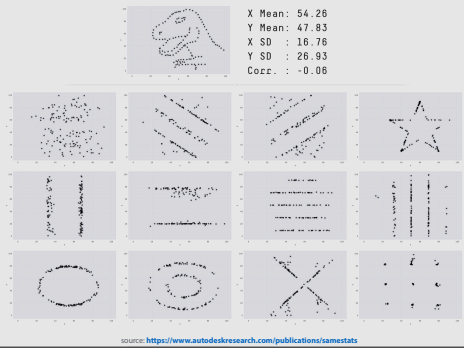
source: By Anscombe.svg: Schutz(label using subscripts): Avenue - Anscombe.svg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9838454

5

One important aspect of not taking the data at face value is to *look* at it.

These are the four datasets from the previous slide. They are a common example, called Anscombe's quartet. Only when we look at the data, do we see how different they are.

More importantly, only when we look at the data, do we see the patterns that define them. These are the patterns we want to get at if we want to understand the data. And none of them are revealed by the descriptive statistics of the previous slides.



source: https://www.autodeskresearch.com/publications/samestats

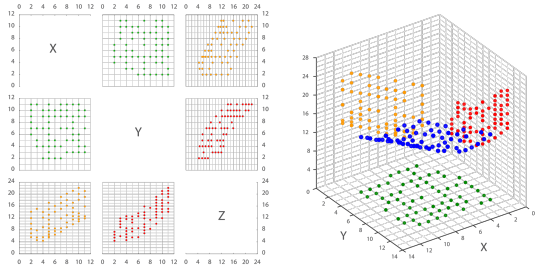
6

Here is a more modern variant: the datasaurus dozen.

Recommended reading: **https://www.autodeskresearch.com/publications/samestats**

---

scatter plot matrix



source: RIDC NeuroMat, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons

7

In machine learning and data science, our datasets are rarely two-dimensional, so we don't have the luxury of simply doing a scatter plot. Looking at our data, in a way that provides insight almost always requires a lot of ingenuity and creativity.

For high-dimensional, multivariate data, of the kind we've been dealing with so far, a good place to start is to produce a **scatter plot matrix**. This is simply a large grid of every scatter plot you can produce between any two features in your data. Often, only the plots below or above the diagonal are shown. The scatterplot matrix gives you a good idea of how the features relate to each other.

If you have a target value (a class or a regression target), it's a good idea to include it among the features for the scatter plot matrix. That way, you can see what relation each feature has with the target in isolation from the other features.

On the right, we see the data as a 3D point cloud (in blue), together with the three projections to 2d (in yellow red and gree) that the scatterplot matrix gives us.

---

cleaning your data

- Missing data:
  missing labels
  missing values
- Outliers
- Class imbalance

8

In the rest of this video, we'll look at ways you can clean up your data, to make it useable for a classification or a regression task.

---

missing values

income	status	unemployed
32000	married	true
	single	false
89000		true
34000	divorced	false
54000	married	true
		false
21000		true
25000	single	true

9

We'll start with missing data. Quite often, your data will look like this.

You will need to do something about those gaps, before any machine learning algorithm will accept this data.

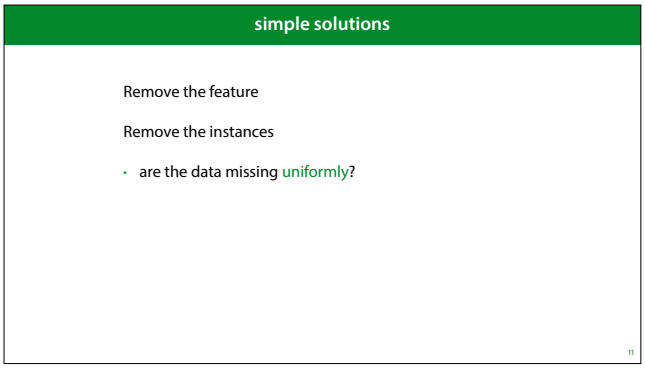
---

missing *labels*

income	status	unemployed
32000	married	true
2000	single	
89000	single	true
34000	divorced	false
54000	married	
34000	married	false
21000	divorced	
25000	single	true

10

What approach you should take is different, depending on whether values from the feature columns are missing, or values from the target column are missing.



If you have missing values in one of your features, the simplest way to get rid of them is to just remove the feature(s) for which there are values missing. If you're lucky, the feature is not important anyway.

You can also remove the instances with missing data. Here you have to be careful. If the data was not corrupted uniformly, removing rows with missing values will change your data distribution.

For example, you might have data gathered by volunteers in the street using some electronic equipment. If the volunteer in Amsterdam had problems with their hardware, then their data will contain missing values, and the collected data will not be representative of Amsterdam.

Another way you might get non-uniformly distributed missing data is if your data comes from a questionnaire , where people sometimes refuse to answer certain questions. For instance, if only rich people refuse to answer questions about their taxes, removing these instances will remove a lot of rich people from your data and give you a different distribution.

How can you tell if data is missing uniformly?
There's no surefire way, but usually you can get a good idea by plotting a histogram of how much data is missing against some other feature. For instance if the number of instance with missing features against income is very different from the regular histogram over income, you can assume that your data was not corrupted uniformly.

Let's zoom out a little before we move on. Whenever you have questions about how to approach something like this, it's best to think about the **real-world setting** where you might apply you trained model. We often call this "production", a term used in software development for the system that will be running the final deployued version of the software. This is where some machine learning models end up, but we might also use machine learning models to perform business analytics, clinical decision support or in a scientific experiment. Wherever your model is meant to end up after you've finished your experiment, that's what we'll call **production**.

And production is what you're trying to simulate, when

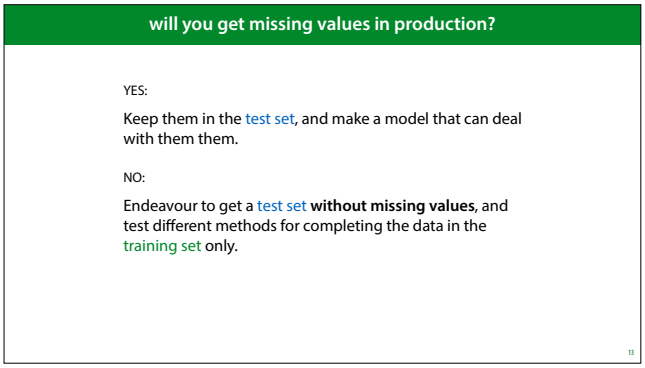
you train your model and test it on a test set. So the choices you make, should make your experiment as close of a *simulation* of your production setting as you can manage.

For example, in the case of missing values, the big question is: can you expect missing datain production? Or, will your production data be clean, and are the test data just noisy because the production environment isn't ready yet.

Examples of production systems that should expect missing data are situations where data comes from a web form with optional values or situations where data is merged from different sources (like online forms and phone surveys).

You may even find yourself in a situation where the test data has not missing values (since it was carefully gathered) but the production system will have missing values (because there the data will come from a web form). In that case, you may want to introduce missing values artificially in your test data, to simulate the production setting and study the effect of missing data.

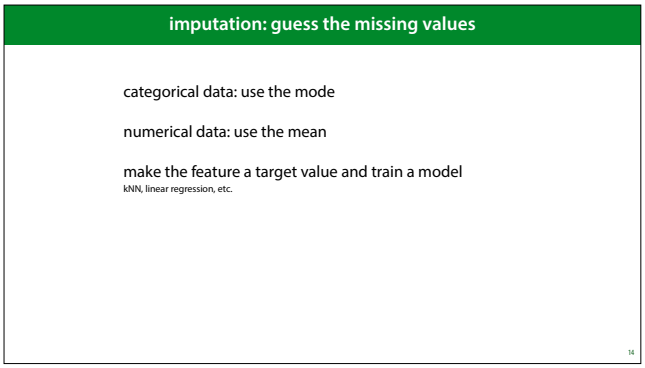
So remember, whenever you're stuck on how to process your data: think what the production setting is that you're trying to simulate, and make your choices based on that.



If you expect to see missing in production, then you model need s to be able to consume missing values, and you should keep them in the test set. For categorical features, the easiest way to do this is to add a category for missing values. For numerical features, we'll see some options in the next slide.

If your production setting won't have missing values, then that's the setting you want to simulate. If at all possible, you should get a test set without missing values, even if the training set has them. You can then freely test what method of dealing with the missing training values gives the best performance on the test set.

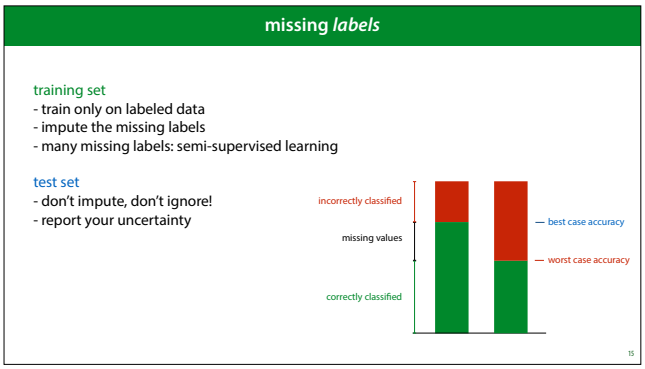
If you cannot get a test set without missing values, one thing you can do is to report performance on both the data that has the instances with missing values removed and the data that has the missing values filled in by some mechanism. Neither are ideal simulations of the production setting, but the combination of both numbers hopefully gives you some idea.



At some point, either in the training data or the test data, we will probably need to fill in the missing values. This is called **imputation**.

A simple way to do this in categorical data is to use the **mode**, the most common category. For numeric data, the **mean** or **median** are simple options. We'll look at when you should use which later in this video.

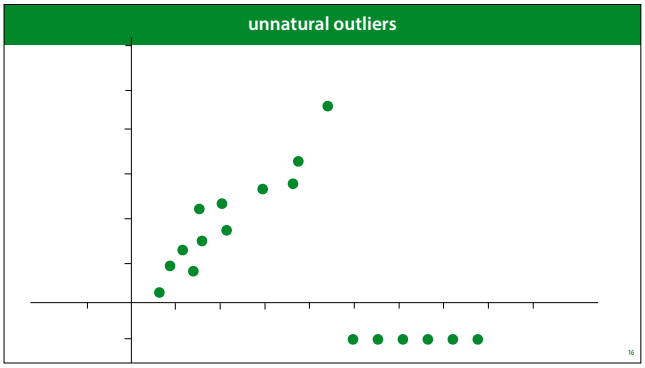
A more involved, but more powerful way, is to **predict the missing value** from the other features. You just turn the feature column in to a target column and train a classifier for categoric features, and a regression model for numeric features.



If your target label has missing values, the story is a little different. In the training set you are free to do whatever you think is best. You can remove instances, or impute the missing labels. If you have a lot of missing labels, you essentially have a semi-supervised learning setting as we saw in the first lecture.

On the test set however, you shouldn't impute or ignore the missing values, since that changes the task, and most likely makes it easier, which will give you false confidence in the performance of your model. Instead, you should report the uncertainty created by the missing values.

In classification, this is easy: you compute the accuracy under the assumption that your classifier gets all missing values correct and under the assumption that it gets all missing values wrong: this gives you a best case and a worst case scenario, respectively. Your true accuracy on the test set is somewhere in between.

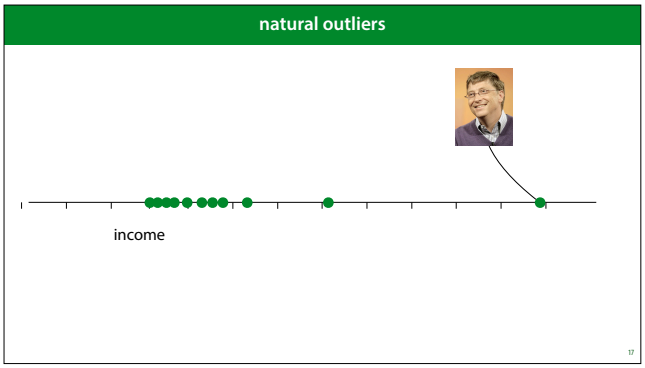


Another problem that we need to worry about, is **outliers**. Values in our data that take on unusual and unexpected values.

Outliers come in different shapes and sizes. The most important question is whether your outliers are natural or unnatural.

Here, the six dots to the right are so oddly, mechanically aligned that we are probably looking at some measurement error. Perhaps someone is using the value -1 for missing data.

We can remove these, or interpret them as missing data, and use the approaches just discussed.

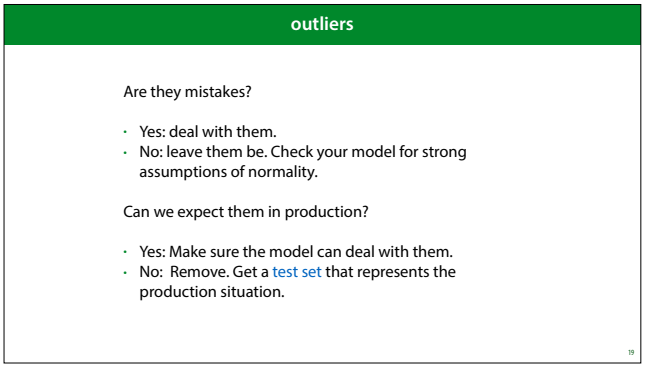


In other cases, however, the "outlier" is very much part of the distribution. This is what we call a natural outlier. Bill Gates may have a million times the net worth of anybody you are likely to meet in the street, but that doesn't mean he isn't part of the distribution of income.

If we fit a normal distribution to this data, the outlier would ruin our fit, but that's because the data *isn't normally distributed*. What we should do is recognize that fact, and adapt our model, for instance by removing assumptions of normally distibuted data.

Here's a metaphor for natural and unnatural outliers. If our instances are image of faces, the image on the left, that of comedian Marty Feldman, is an extreme of our data distribution. It looks unusual, but it's crucial in fitting a model to this dataset. The image on the right is clearly corrupted data. It tells us nothing about what human faces might look like, and we're better off removing it from the data.

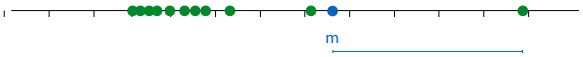
However, remember the real-world use-case: if we can expect corrupted data in production as well, then we should either train the model to deal with it, or clean it automatically in production as well. This would require us to have some way to detect automatically, whether something is an outlier. If the outliers are rare, and we have a lot of data, it may be easier just to leave them in and hope the model can learn to work around them, even if they are unnatural outliers.



If you have very extreme values that are not mistakes (like Bill Gates earlier), your data is probably not normally distributed. If you use a model which assumes normally distributed data, it will be very sensitive to these kinds of "outliers". It may be a good idea to remove this assumption from your model (or replace it by an assumption of a heavy-tailed distribution).

Note that you have to know your model really well to know if there are assumptions of normality. For instance anything that uses *squared errors* essentially has an assumption of normality.

fitting a central value



$$\arg\min_{m} \sum_{i} (m - x_i)^2$$

To illustrate: let's learn which *single value* best represents our data. We choose a value m, compute the distance to all our data points (the residuals) and try to minimise their squares. This is a one dimensional version of the linear regression. The only assumption we've made is that of **squared errors**.

$$\arg\min_{m} \sum_{i} (m - x_i)^2$$

$$\sum_{i} \frac{\partial (m - x_i)^2}{\partial m} = 0$$

$$\sum_{i} \frac{\partial (m - x_i)^2}{\partial m - x_i} \frac{\partial m - x_i}{\partial m} = 0$$

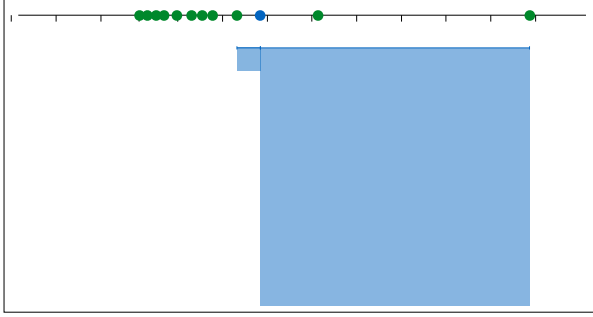
$$2 \sum_{i} m - x_i = 0$$

$$nm - \sum_{i} x_i = 0$$

$$m = \frac{\sum_i x_i}{n}$$

We take the derivative of the objective function and set it equal to zero.No gradient descent required here, we'll just solve the problem analytically.

What we find is that the optimum is **the mean**. The assumption of squared errors leads directly to the use of the mean as a *representative example* of a set of points.



We can now see why the the assumption of squared errors is so disastrous in the case of the income distribution.

If Bill Gates makes a million times as much as the next person in the dataset, he is not pulling on the mean a million times as much, he's pulling 10^{12} times as much. The inclusion

Hence the joke: A billionaire walks into a homeless shelter and says "What a bunch of freeloaders, the average wealth in this place is more than a million dollars!"

---

mean absolute error

$$\arg\min_{m} \sum_{i} |m - x_i|$$

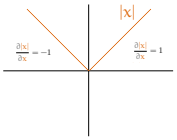
To get rid of the normality assumption, or rather, replace it by another assumption, we can use the **mean absolute error** instead. We take the residuals, but we sum their *absolute value* instead of their *squared value*. Which is the most representative value that minimises *that* error?

$$\arg\min_m \sum_i |m - x_i|$$

$$\sum_i \frac{\partial |m - x_i|}{\partial m} = 0$$

$$\sum_i \frac{\partial |m - x_i|}{\partial m - x_i} \frac{\partial m - x_i}{\partial m} = 0$$

$$\sum_i \text{sign}(m - x_i) = 0$$



24

To work this out, we need to know the derivative of the absolute function. This function is the identity if the argument is positive (so its derivative is 1) and the negative identity if the argument is negative

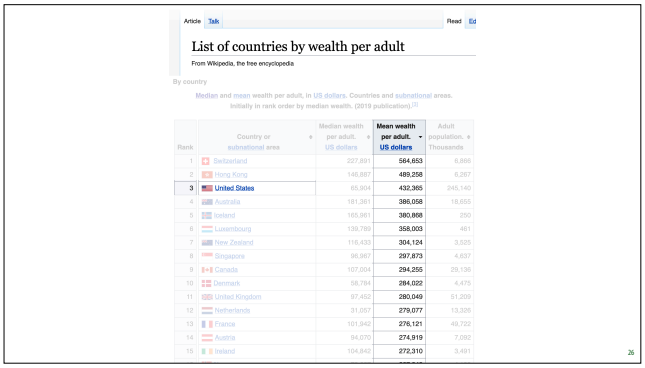
---

mean absolute error -> median

$$\sum_i \text{sign}(m - x_i) = 0$$

25

We've worked out that the value m that minimizes this error is the one for which the signs of the residuals sum to zero. This happens if the sum contains as many "-1"s as "+1"s, that is, if we have as many instances to the left of m as we have to the right.

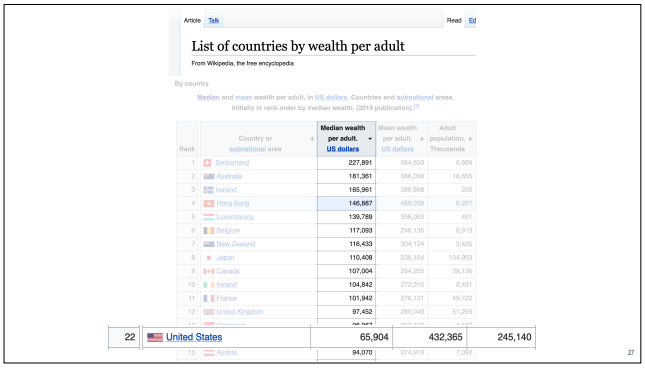


26

This mistake, of using the mean when a normal distribution is not an appropriate assumption, is sadly very common.

For example, you might hear someone say something like "there's no poverty in the US, it's the third richest country in the world by average personal wealth".

Wikipedia allows us to fact-check this quickly and it is indeed true. But remember that Bill Gates and Jeff Bezos live in the US, and as we saw, they have a pertty strong pull on the mean. Luckily, Wikipedia also allows us to sort the same list by *median* wealth.



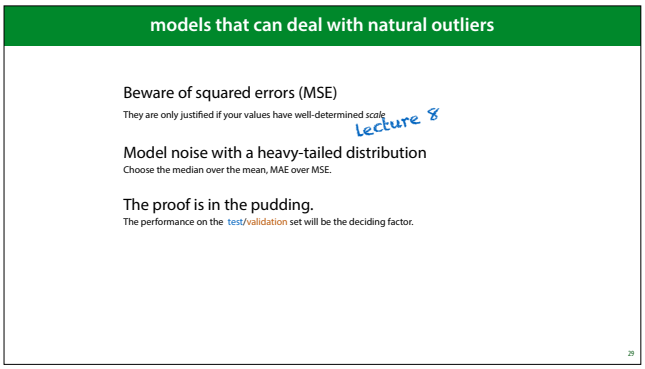
27

If we do that, we see that the US suddenly drops to 22nd place.

The Netherlands drops from 12th to 34, incidentally. So there's plenty of income inequality over here as well.

Here's an example of this fallacy in the wild. In 2019, there was a discussion in the US about unionisation in the games industry. Here, one of the heads of Take-Two suggests that because the *average* yearly salary has six figures, unions are unlikely.

Whether rich people can benefit from unions is a question for a different series of lectures, but the fact that the average wages are high, most likely just means that there is a small number of very rich people in the industry. We'd need to know the *median* to be sure.

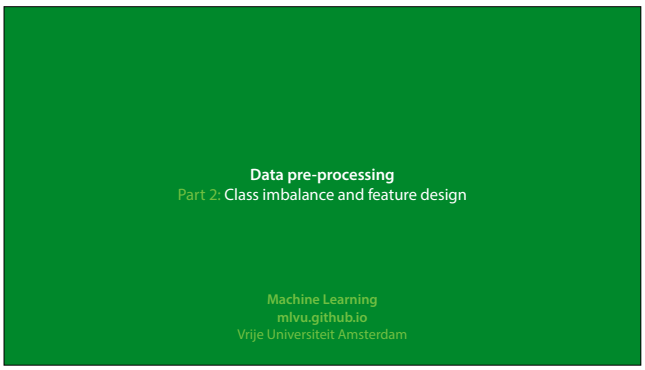


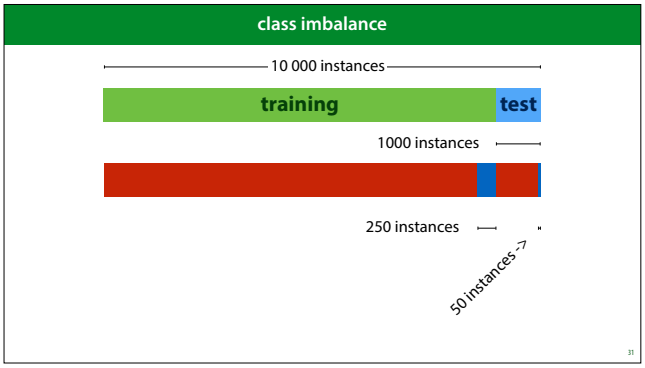
If you want to adapt your model to deal with natural outliers, beware of hidden assumptions of normality.

Consider modelling your noise with a heavy-tailed distribution instead, in other words, one which makes outliers more likely. Using the median instead of the mean is one way to do this.

If you are doing regression and your target label is non-normally distributed then you can use the sum of absolute errors as a loss function instead of the sum of squared errors. This will also implicitly assume a more heavy-tailed distribution than the normal, but even more heavy tailed distributions are available.,

---





In the last lecture, we saw that class imbalance can be a big problem. We know what we can do to help our analysis of imbalanced problems, but how do we actually improve training?

class imbalance

Use a big test set.

Don't rely on accuracy. Try ROC plots, precision-recall plots, AUC, etc. Look at the confusion matrix.

Resample your training data.

Use data augmentation for the minority class

32

---

oversampling

33

The most common approach is to oversample your minority class by sampling with replacements. The advantage is that this leads to more data. The disadvantage is that you end up with duplicates in you dataset. This may increase the likelihood of overfitting.

---

undersampling

34

You can also under sample your majority class. This doesn't lead to duplicates, but it does mean you're throwing away data.

If you have use an algorithm that makes multiple passes over the dataset (like stochastic gradient descent, explained next week) it can help to resample the dataset fresh for every pass.

---

data augmentation

SMOTE: add midpoints between nearby minority class data points.

Domain specific: rotate or translate images in the minority class. Add Gaussian noise.

Remember: only on the training data. Keep the test data as is.

more information: https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets

35

A more sophisticated approach is to oversample the minority class with new data derived from the existing data.

SMOTE is a good example: it finds small clusters of points in the minority class, and generate their mean as a new minority class point. This way, the new point is not a duplicate of any existing point, but it is still in a region that contains a lot of points in the minority class.

We don't have time to go into this deeply. If you run into this problem in your project, click the link for a better explanation.

getting features

phone nr	income	status	unemployed	birthdate	age
0646785910	32000	married	true	4-5-78	41
0207859461	45000	single	false	3-6-00	19
0218945958	89000	married	true	4-7-91	28
0645789384	34000	divorced	false	3-11-94	25
0652438904	54000	married	true	21-3-95	24
0309897969	36000	single	false	4-12-46	73
0159874645	21000	single	true	13-8-52	67

Even if your data comes in a table, that doesn't necessary mean that every column can be used as a feature right away (or that this would be a good approach).

---

from: date, phone number, images, status, text, category, tags, etc…

to: numeric, categoric, both.

Some algorithms (like linear models or kNN) work only on numeric features. Some work only on categorical features, and some can accept a mix of both (like decision trees). Translating your raw data into features is more an art than a science, and the ultimate test is the test set performance. But let's look at a few examples, to get a general sense of the way of thinking.

---

age

age
41
19
28
25
24
73
67

to numeric: From integer to real-valued. Not usually an issue.

to categoric: Group data into bins? E.g. above or below the median.

- Information loss is unavoidable.

Age is integer valued, while numeric features are usually real-valued. In this case, we can just interpret the age as a real-valued number, and most algorithms won't be affected.

If our algorithm only accepts categoric features, we'll have to group the data into bins. For instance, you can turn the data into a two-category feature with the categories "below the median" and "above the median".

We'll lose information this way, which is unavoidable, but if you have a classifier that only consumes categorical features, and works really well on your data, it may be worth it.

---

phone number

phone nr
0646785910
0207859461
0218945958
0645789384
0652438904
0309897969
0159874645

to numeric: From integer (?) to real-valued. Highly problematic.

to categoric: area codes, cell phone vs. landline

We can represent phone numbers as integers too, so you might think the translation to numeric is fine. But here it makes no sense at all. Translating to a real valued feature would impose an *ordering* on the phone numbers that would be totally meaningless. My phone number may represent a higher number than yours, but that has no bearing on any possible target value.

What *is* potentially useful information, is the area code. This tells us where a person lives, which gives an indication of their age, their political leanings, their income, etc. Wether or not the phone number is for a mobile or a landline may also be useful. But these are **categorical features**.

categoric to numeric

integer coding:

genre	genre
sci-fi	1
romance	2
comedy	4
thriller	3
thriller	3
romance	2
romance	2
sci-fi	1
thriller	3
comedy	4

one-hot coding:

scifi	romance	comedy	thriller
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
0	0	0	1
0	1	0	0
0	1	0	0
1	0	0	0
0	0	0	1
0	0	1	0

aka 1-of-N coding

40

So what if our model only accepts numerical features? This is very common: most modern machine learning algorithms are purely numeric. How do we feed it categorical data? Here are two approaches.

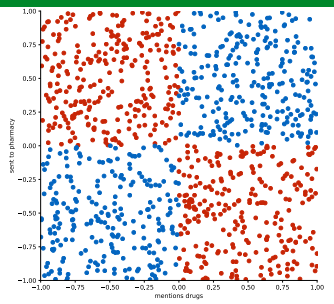
Integer coding gives us the same problem we had earlier. We are imposing a false ordering on unordered data.

One-hot coding avoids this issue, by turning *one* categorical feature into *several* numeric features. Per genre we can say whether it applies to the instance or not.

In general, the one-hot coding approach is preferable.

---

expanding features



41

Once we've turned all our features into data that our model can handle, we can still manipulate the data further, to improve performance.

How to get the useful information from your data into your classifier depends entirely on what your classifier can handle. The linear classifier is a good example. It's quite limited in what kinds of relations it can represent. Essentially, each feature can only influence the classification boundary in a simple way. It can push it up or down, but it can't let its influence depend on the values of the other features. Here is a (slightly contrived) example of when that might be necessary.

Imagine classifying spam emails on two features: to what extent the email mentions drugs, and to what extent the email is sent to a pharmaceutical company. This problem is completely impossible for a linear classifier.

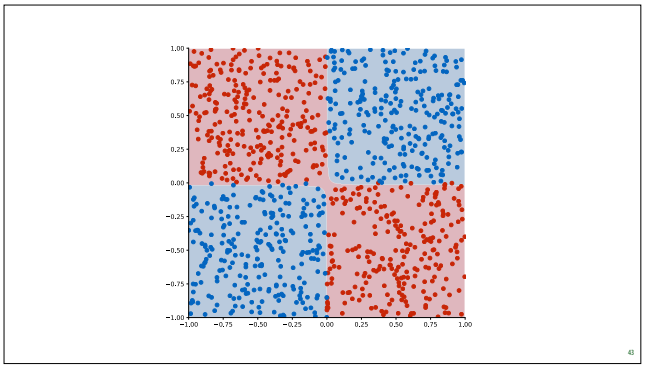
---

cross product

d	p	d * p
0.75	0.98	0.74
-0.66	-0.32	0.21
-0.45	0.84	-0.38
0.93	0.78	0.72
-0.42	0.24	-0.10
-0.02	0.43	-0.01
-0.74	0.58	-0.43
-0.41	-0.41	0.17
0.59	0.72	0.42

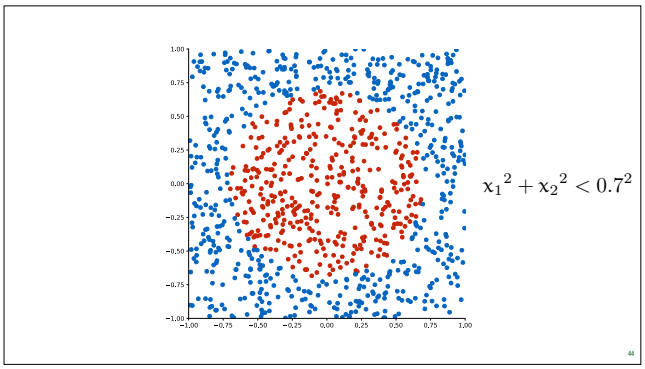
42

We can switch to a more powerful model, but we can also add power to the linear classifier by **adding extra features derived from the existing features**. Here, we've added the cross-product of d and p. (Note the XOR relationship of the signs: two negatives or two positives both make positive, a negative and a positive make a negative). Now the classifier can separate the classes perfectly by just looking at whether or not the third column is positive or negative.

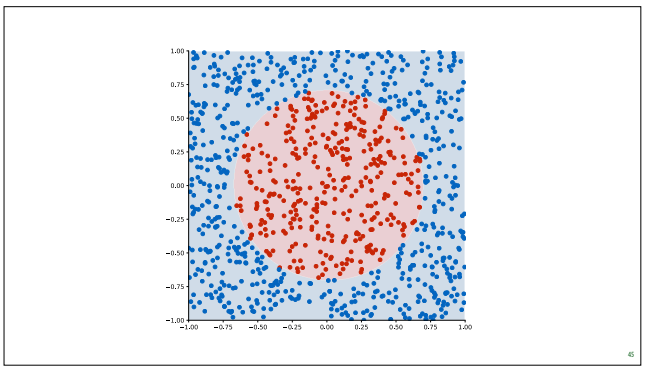


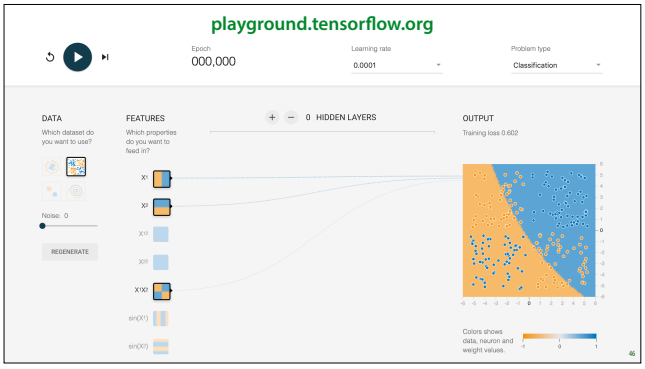
This is a linear classier that operates in a 3D space. But since the third dimension is derived from the other two, we can colour our original 3D space with the classifications. Projected down to 2D, the classifier solves our XOR problem perfectly.

You can try this yourself at **playground.tensorflow.org**.



One more example. Here we color points red if the distance to the origin is less than 0.7. Again, this dataset is not at all linearly separable. Using Pythagoras, however, we can express how the classes are decided: if $x_1^2 + x_2^2 < 0.7^2$ then we classify as red, otherwise as blue. This is a linear decision boundary for the features x_1^2 and x_2^2.

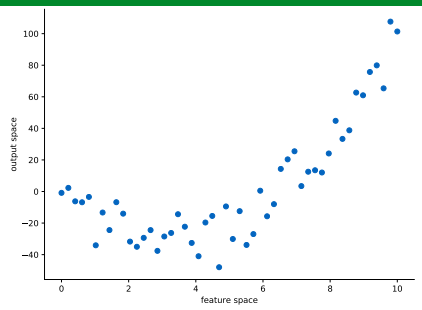




You can test this idea in the tensorflow playground. Just turn on some of the extra features in the `features' column.

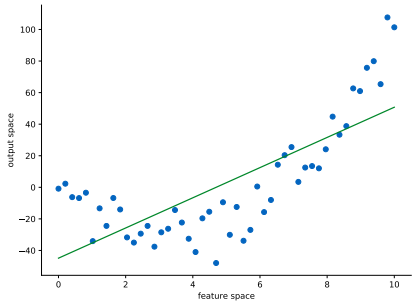
https://bit.ly/2SI3Krp

regression



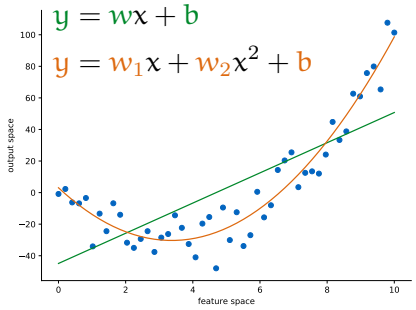
47

We can do the same thing with regression. Here, we have a very non-linear relation.



48

A purely linear classifier does a terrible job.



49

We *can* fit a parabola through the data perfectly. We can see this as a more powerful model, but we can also see this as a 2D linear regression problem, where the second feature (x^2) is derived from the first.

This is relevant because linear models are extremely simple to fit. By adding derived features we can have our cake and eat it too. A *simple* model that we can fit quickly and accurately, and a *powerful* model that can fit many nonlinear aspects of the data.

If we don't have any intuition for which extra features might be worth adding, **we can just add all cross products**. Other functions like the sine or the logarithm may also help a lot.

---

adding features

Can make a weak classifier (especially a linear one) stronger.

Any function on one or more of the existing features can work.

The model stays *convex*: easy to solve, optimal solution guaranteed.

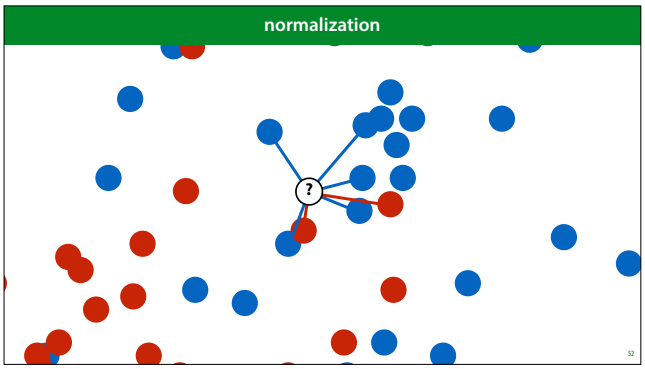
Common choice: all 2-way cross products, all 3-way cross products, etc.

$x, y, z \rightarrow x, y, z, xy, zy, xy, x^2, y^2, z^2, xyz, x^2y, \ldots, x^3, \ldots$

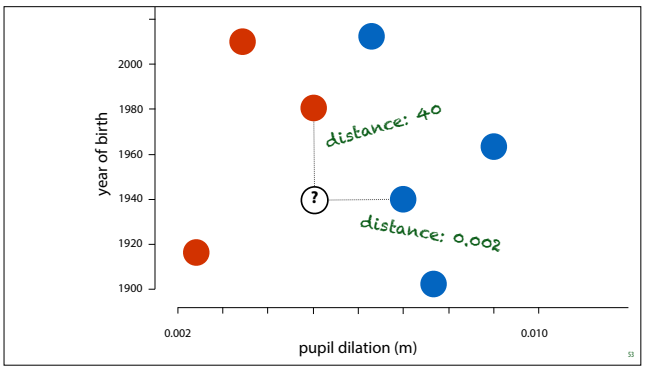
50

Data pre-processing
Part 3: Normalization

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam



For some models, it's important to make sure that all numeric features have broadly the same minimum and maximum. In other words, that they are *normalized*. To see why, let's go back to the kNN classifier from the first lecture.

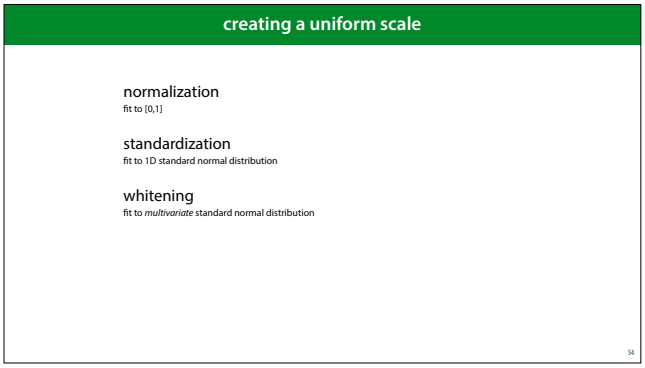


Imagine we are using a 1-NN classifier (i.e. it only looks at the nearest example, and copies its class).

In this plot, it looks like the blue and the red dot are the same distance away.

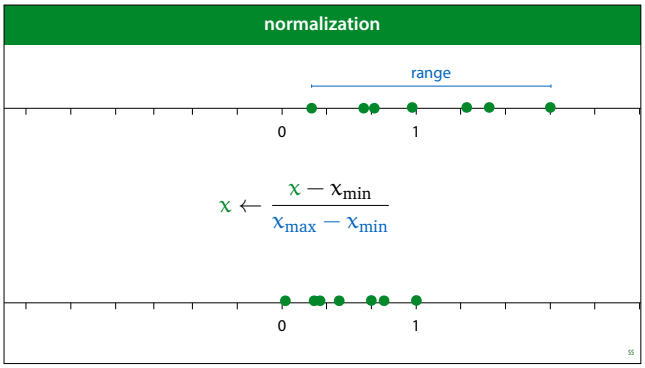
But note the range of values for the two features: years and pupil dilation. Because years are measured in bigger units than than pupils, the blue dot will always be much closer. But this distinction is not meaningful. What we want to look at is how much spread there is *in the data*, and use that as our distance.

We do that by normalizing our data before feeding it to the model.



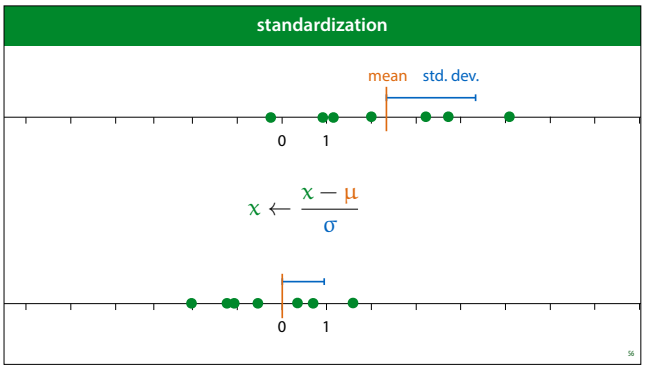
We'll discuss three approaches to solving this problem. Normalization, which reshapes all values to lie within the range [0, 1], standardization, which reshapes the data so that its mean and variance are those of a standard normal distribution (0 and 1 respectively) and whitening, which looks at features *together*, to make sure that as a whole their statistics are those of a multivariate standard normal distribution.

These terms are often used interchangeably. We'll stick to these definitions for this course, but in other contexts you should check that they mean what you think they mean.

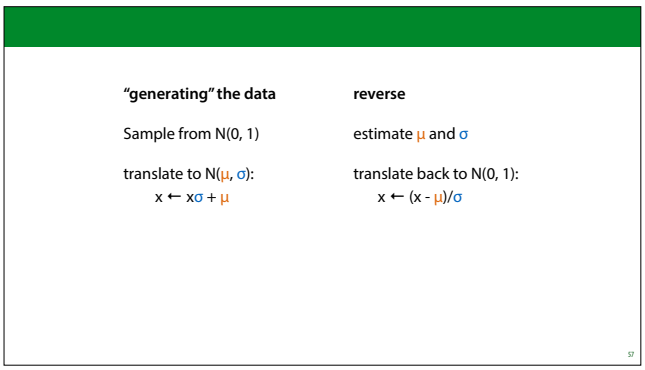


Normalisation scales the data linearly so that the smallest point becomes 0 and the largest becomes 1. Note that because x_{min} is negative (in this example), we are actually moving all data to the right, and then rescaling it.

We do this independently for each feature.



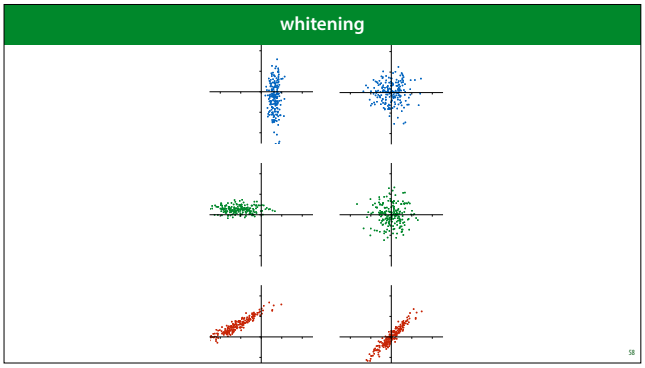
Another option is standardization. We rescale the data so that the mean becomes zero, and the standard deviation becomes 1. In essence, we are transforming our data so that it looks like it was sampled from a standard normal distribution (or as much as we can with a one dimensional linear transformation).



We can think of the data as being generated from a standard normal distribution, followed by multiplication by sigma, and and adding mu. The result is the distribution of the data.

If we then compute the mean and the standard deviation of the data, the formula in the slide is essentially inverting the transformation, recovering the "original" data as sampled from the standard normal distribution.

We will build on this perspective several times throughout the course.

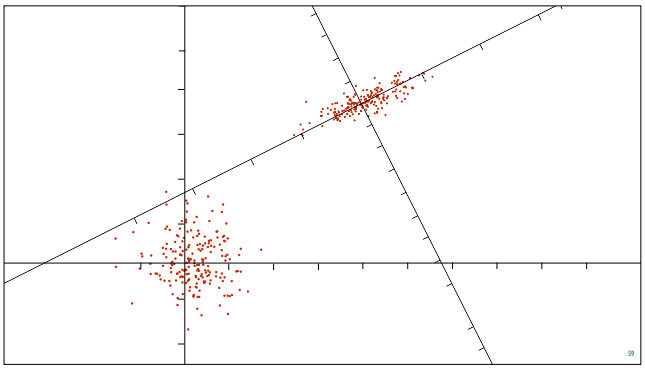


Here's what standardization looks like if we apply it to data with two features. If the data is *uncorrelated*, we are reducing it to a nice spherical distribution, centered on the origin, with the same variance in each direction. Exactly what data from a **multivariate standard normal distribution** looks like.

If, however, our data is correlated, that is; knowing the value of one feature helps us predict the value of the other, we get a different result. This is because we standardize each feature *independently*, and the features are not independent. Is there a way to achieve the same effect with the correlated data? Can we transform the features somehow so that it looks like they came from a distribution like the one top right?

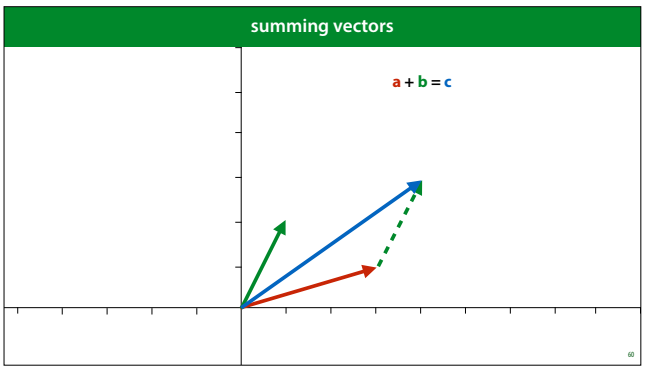
This is what **whitening** can do for us.

Note that this is not usually necessary in practice. Normalizing or standardising each feature independently is usually fine, especially if the your model is powerful enough to learn correlations**.** However, for the rest of the lecture, it's instructive to see how to perform this transformation.

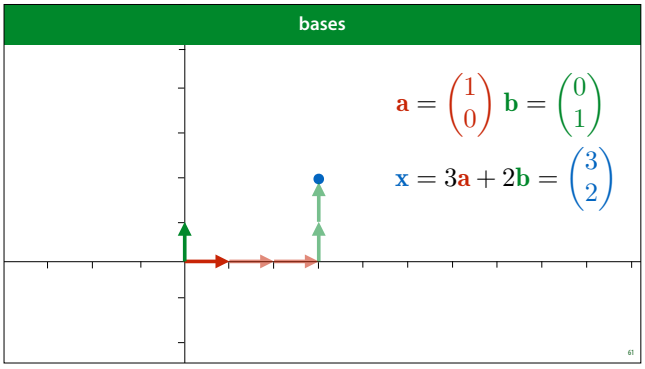


In essence we want to transform the data top right to something that looks like the data bottom left. Or, the same question asked differently, can we express the data in another **coordinate system**, to that in the new coordinate system, the features are not correlated and the variance in the direction of each axis is 1?

In order to show how to do this we need to revise some bits of linear algebra. Specifically, we need to look at **linear bases** (the plural of basis).



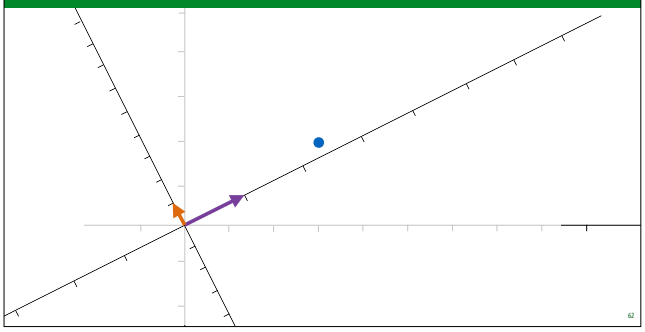
Here's a quick reminder of how summing vectors works. We stick the tail of b onto the head of a and draw a line from the tail of a to the head of b.



We can see our basic Cartesian coordinate system as made up entirely of the two vectors (1 0) and (0 1). To describe a point in the place, we just sum a number of copies of these vectors.

Every point in the plane is just a linear combination of these two. A coordinate like (3, 2) means: "sum three copies of b_1 and add them to two copies of b_2." We call these **basis vectors**: vectors that allow us to describe all points in a space in terms of a multiple of each of the basis vectors. The set of points that can be described in this way is the space **spanned** by the basis vectors.

bases



If we choose different **basis vectors**, we get a different coordinate system to express our data in. But (excepting some rare choice of basis vectors), we can still express all the same points as a number of copies of one vector, plus a number of copies of the other.

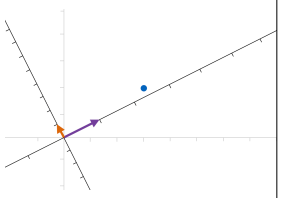
---

bases

$$\mathbf{c} = \begin{pmatrix} 1.26 \\ 0.9 \end{pmatrix} \quad \mathbf{d} = \begin{pmatrix} -0.3 \\ 0.5 \end{pmatrix}$$

$$\mathbf{x}_s = 2.5\mathbf{c} + 0.5\mathbf{d} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

$$\mathbf{x}_b = \begin{pmatrix} 2.5 \\ 0.5 \end{pmatrix}$$



If we know the coordinates $\mathbf{x}_b$ in our non-standard coordinate system, it's easy to find the standard coordinates $\mathbf{x}_s$. We just multiply the first coordinate of x_b with the first basis vector, the second coordinate with the second basis vector and sum the result.

---

rebasing

$$\mathbf{B} = [\mathbf{c}, \mathbf{d}] = \begin{bmatrix} 1.26 & -0.3 \\ 0.9 & 0.5 \end{bmatrix}$$

$$\mathbf{x}_s = \mathbf{B}\mathbf{x}_b \qquad \mathbf{x}_s = 2.5\mathbf{c} + 0.5\mathbf{d} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

$$\mathbf{x}_b = \mathbf{B}^{-1}\mathbf{x}_s$$

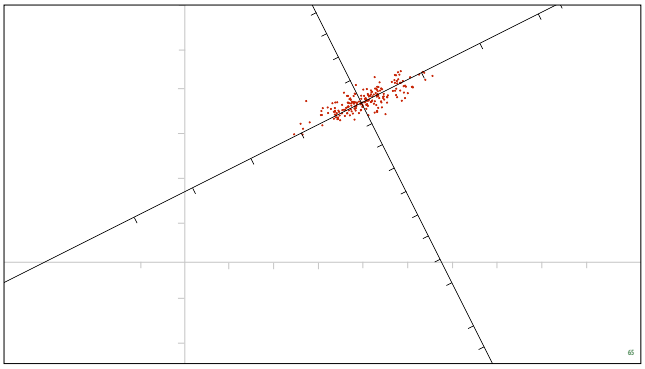
If the columns of **B** are orthogonal unit vectors, then **B** represents an *orthonormal* basis. In that case:

$$\mathbf{x}_b = \mathbf{B}^{\mathsf{T}}\mathbf{x}_s$$

The basis vectors are usually expressed as the columns of a matrix **B**. That way, transforming a coordinate **x** in basis B to the standard coordinates can be done simply by matrix multiplying **B** by **x**. It also tells us that to go the other way, to transform a standard coordinate to the basis, you multiply by the *inverse* of **B**.

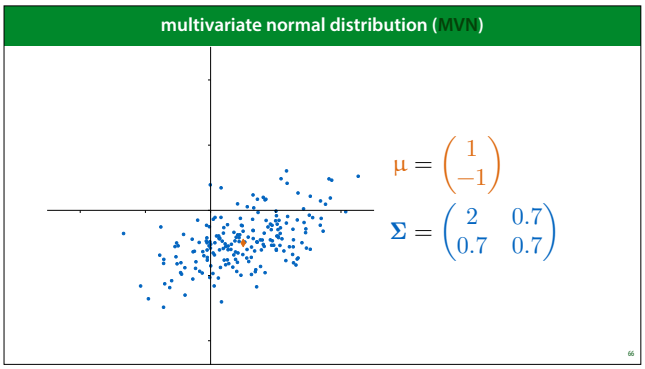
Since inverting a matrix is an expensive and numerically unstable business, it's good to focus (if possible) on **orthonormal bases**. That is, bases for which the basis vectors are **orthogonal** (the angle between any two bases is 90 degrees) and **normal** (all vectors have length 1). In that case the matrix transpose (which is simple to compute without loss of precision) is equal to the matrix inverse, so we can switch back and forth between bases quickly, without losing information.

Here [a,b] represents the matrix created by concatenating the vectors a and b.



We can now re-phrase what we're aiming to do: we want to find a set of new *basis vectors* so that we can express the data in a coordinate system where the features are not correlated, and the variance is 1 in every direction.

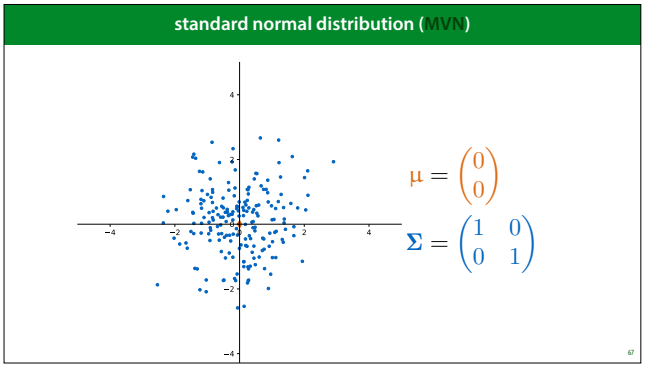
Note that the latter means we can't have an orthonormal basis (the basis vectors can't be one). There are some tricks to deal with this, but we'll not mention them here.



To figure out how to find this basis, we will follow the same principle as we did with standardisation: we will assume that the data was generated by a standard multivariate normal distribution (MVN), followed by a translation and a change of basis (with the change of basis causing some features to become correlated). We will attempt to reverse the process by:

- fit a (nonstandard) MVN to the data
- figure out the transformation that transforms the standard MVN to this MVN
- apply the inverse of this transformation.

A multivariate normal distribution is a generalisation of a one-dimensional normal distribution. Its mean is a single point, and its variance is determined by a symmetric matrix called a covariance matrix. The values on the diagonal indicate how much variance there is along each dimension. The off-diagonal elements indicate how much co-variance there is between dimensions.



The *standard* MVN has its mean at the origin and the identity matrix as its covariance matrix (i.e. its features are uncorrelated, and the variance is 1 along every dimension).

fitting an MVN to data

$$\mathbf{m} = \frac{1}{n}\sum_i \mathbf{x}_i$$
$$\mathbf{X} = \left[\mathbf{x}_1, \ldots, \mathbf{x}_n\right] - \mathbf{m}$$
$$\mathbf{S} = \frac{1}{n-1}\mathbf{X}\mathbf{X}^T$$

The estimators for the **sample mean** and **sample covariance** look like this. Computing these values lets you fit an MVN to your data.

MVNs as transformations

Start with N(**0**, **I**) <-- standard normal

- Let X ~ N(**0**, I)
- Let Y = **A**X + **t**

then:

Y ~ N(**t**, **AA**^{T})

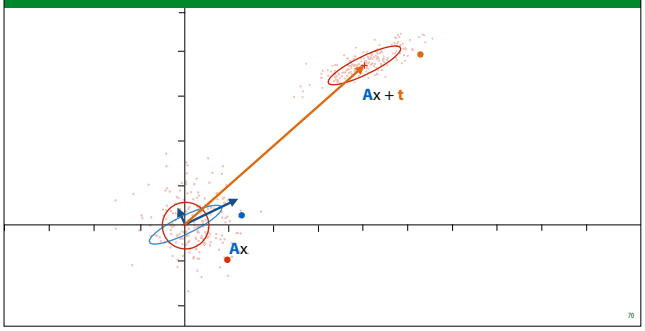
As before, we will imagine starting with a standard MVN, and then transforming our data linearly.

We can *sample* from an n-dimensional **Standard MVN** (with mean at the origin and the identity matrix I as a covariance matrix) by simply filling a length n vector with values sampled from a one-dimensional standard normal distribution.

If we then transform x by multiplying it by some matrix A and adding some vector t, the result is the same as sampling from an MVN with mean **t** and covariance **AA**^{T}.

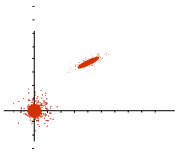
Any MVN can be described in this way as a transformation of the standard normal distribution.

MVNs as transformations



Here's what that looks like. For our data. We imagine some data sampled from a standard MVN. We multiply by some some matrix **A** to squish and rotate it. And then we apply a translation vector **t** to translate it to the right point in space.

whitening: invert this transformation



- Compute **S**, **m** from data.
- Find some **A** such that **S** = **AA**^{T}
  - Cholesky decomposition
  - Singular Value Decomposition (PCA whitening)
  - Matrix square root
- Whiten the data:

$$\mathbf{x} \leftarrow \mathbf{A}^{-1}(\mathbf{x} - \mathbf{m})$$

Of course, we want to do this the other way around: we start with our data, we figure out what A and t are, and then we apply the inverse of the linear operation.

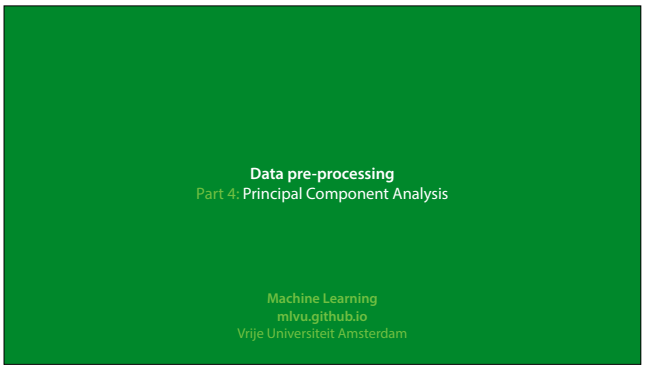
In slide 69, we saw that the covariance after our transformation was **AA**^{T}, so if we estimate the covariance **S** and find some matrix **A** such that **AA**^{T} = **S**, we can then use that A for the inverse transformation. Finding this **A** can be done in many ways. The most stable one is the Singular Value Decomposition, which leads to a method known as PCA whitening, discussed in the next video.

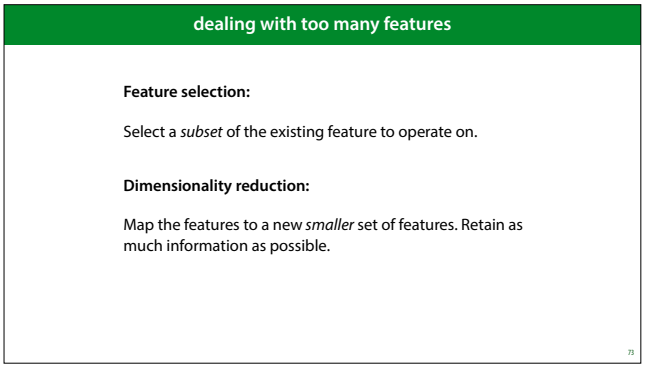
Since the multiplication by **A** doesn't change the mean, we know that the translation vector **t** is equal to the

mean **m**.

Once we know **A** and **t**, we can reverse the transformation as shown here.

Compare this to the standardisation operation: there, we subtract the mean, and multiply by the inverse of the standard deviation. Here we do the same, but in multiple dimensions (note that the standard deviation is the square root of the variance, just like the A matrix squared is the *co*variance).

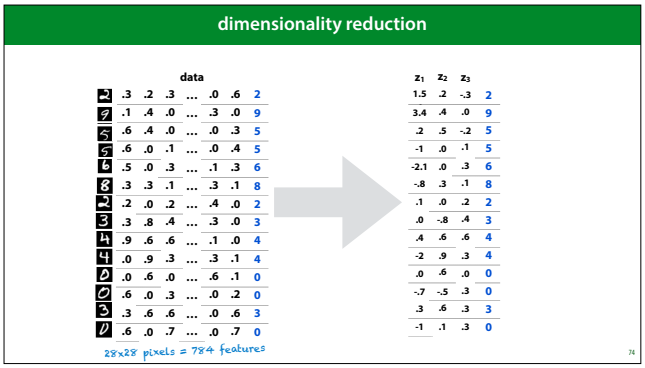




Some datasets have more features than a given model can handle. In that case, there are two things we can do: we can try to find a subset of the features that is most informative, and operate on those. This has the benefit that the features retain their meaning and are still interpretable. This is called **feature selection**.

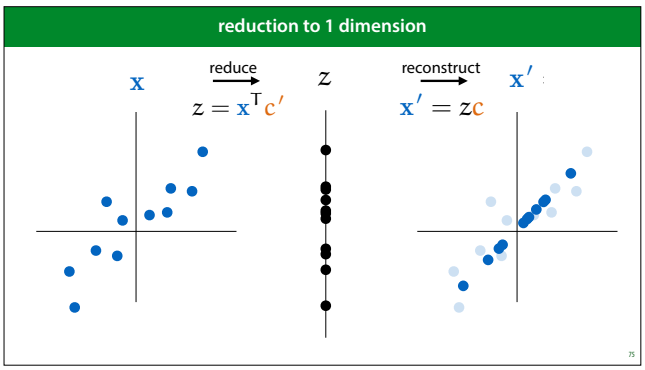
The alternative is to take information from all *all* features and to map them to a new (smaller) set of derived features, which retain as much of the original information as possible. This is called **dimensionality reduction**. In this case, the new features don't always have an obvious meaning, but they may still work well for machine learning purposes.

In this video we will just look at one dimensionality reduction method: Principal Component Analysis (PCA).



Dimensionality reduction is the opposite of the feature expansion trick we saw earlier. It describes methods that reduce the number of features in our data (the dimension) by deriving new features from the old ones, hopefully in such a way that we capture all the essential information. There are several reasons you might want to reduce the dimensionality of your data:

- Efficiency. Many machine learning methods can only handle so many features. If you have a very high dimensional dataset, you may be forced to do some dimensionality reduction in order to be able to run your chosen model.
- Reduce variance of the model performance (make the bias/variance tradeoff). Feature expansion boosts the power of your model, likely giving it higher variance and lower bias. Dimensionality reduction does the opposite: it reduces the power of your model likely giving you higher bias and lower variance.
- Visualization. If you're lucky (or if you have a very strong dim. reduction method), reducing down to just 2 or 3 dimensions preserves the important information in your data. If so, you can do a scatterplot, and use the power of your visual cortex to analyse your data (i.e. you can look at it).

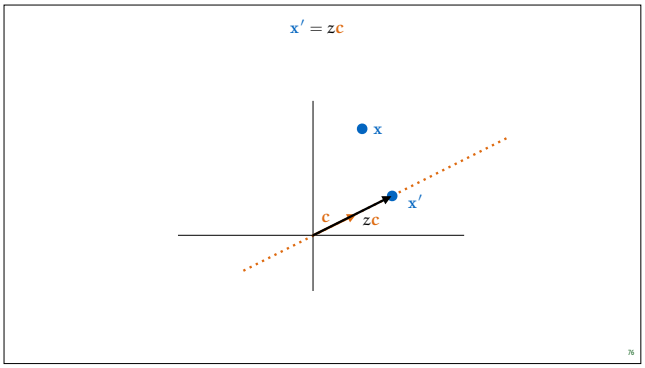


To simplify our explanation, we'll start by reducing our data to just one dimension: for each instance, we'll try to come up with a single number z, that hopefully represents is pretty well.

The way we'll do this is by optimizing the **reconstruction error**. We'll come up with some function that reconstructs our data from the reduced points {z}. The closer this reconstruction is to the original point, the better.

We'll add the constraint that both the function that reduces the data and the function that reconstructs the data should be **linear**. We'll also assume that the data is **mean-centered**, so that we won't need to apply any translations: the mean of the original data, the reduced data, and the reconstructed data is zero or the zero vector.

Under these constriants, the reduction function consist of taking the dot product of our vector with some parameter vector **c'**, and the reconstruction function consists of multiplying our reduced representation with some other parameter vector **c**.

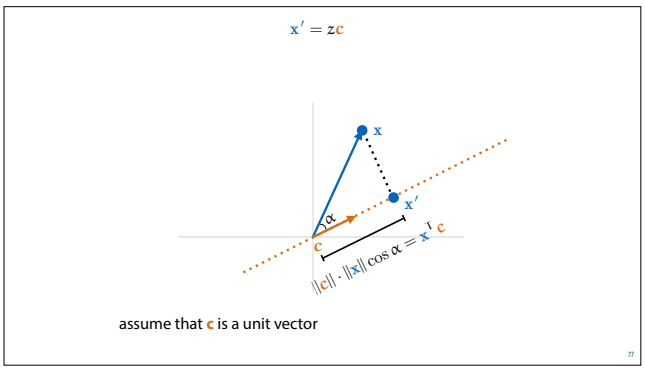


Here we have a diagram for a single instance. Since each reconstruction is a multiple of **c**, the line that the reconstructions lie on, is the line that the vector **c** points along.

We'll work out what our functions should be in the following order. First, we will assume that we have the reconstruction function, and ask what the best reduction function is to use, in terms of that reconstruction function. Then we will work out an optimization objective for both of them together.

So, let's assume that we have areconstruction function. That is, we have **c** already. What value should we then choose for z to give us the best reconstruction?

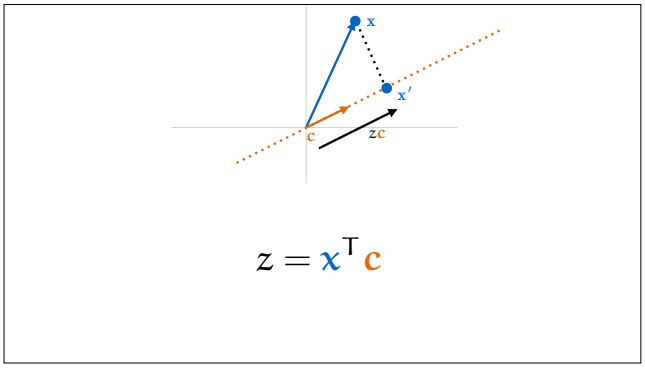
The closest we can get **x'** to **x** is to put **x'** where the line between **x** and **x'** makes a right angle with the line of **c**. This is the **projection** of **x** onto **c**, and if you know your linear algebra, you'll know the length of z**c** in this picture is related to the dot product of **x** and **c**. Why?



For our purposes, the length of **c** doesn't matter (if we make **c** longer or shorter it still defines the same line), so we'll assume that **c** has length 1 (that is, it is a **unit vector**).

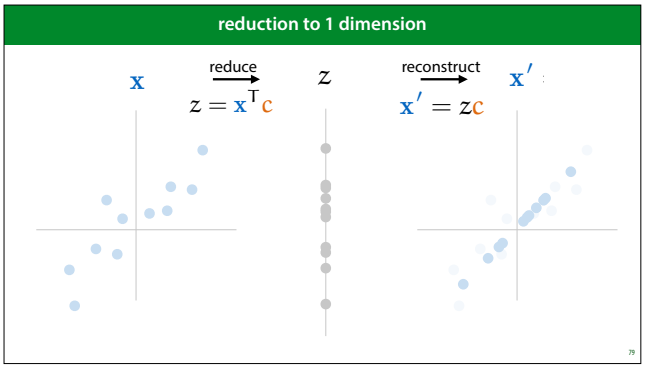
From basic trigonometry, we know that the length of the black line is ||**x**|| cos α. Because ||**c**|| = 1, we can multiply by that without changing the value, which means that the length of the black line is equal to the dot product between **x** and **c**.

See **peterbloem.nl/blog/pca** for a more intuitive proof.

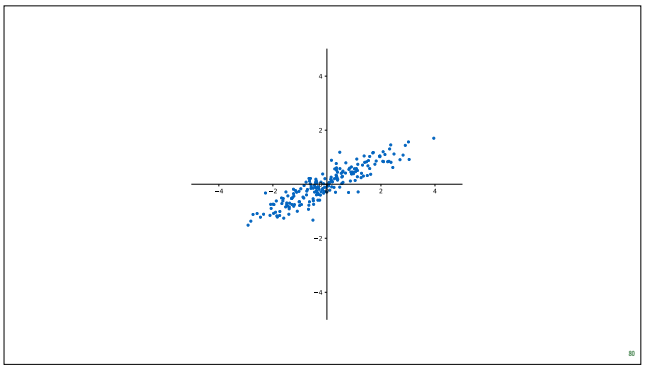


What this tells us, is that the projection of **x** onto **c** is found by taking their dot product. Since **c** has length one, this is the value that we want to multiply **c** by to get to **x'**.

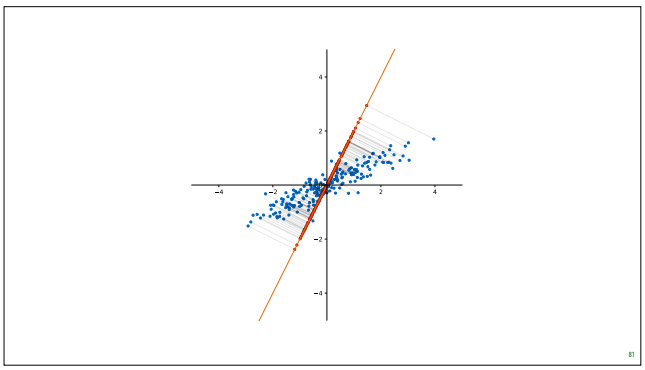
In other words, given **c**, we now know how to find our reduced data z. The vectors in the reduction function and the reconstruction function are the same.



Here's how we've simplified our picture. The reduction and and reconstruction now have the same parameters **c**.



The next question is how to find this **c** for some given data. Before we move on, let's pick an arbitrary **c** and see what we get when we project down onto that vector.



Here it is. The red points are our reconstructions. For each point, the new feature z is the distance from the origin to the red point. The grey lines indicate how far the reconstruction is from the original data.

Clearly, this is not a very good choice for **c**. The grey lines could be much shorter. This is how we'll optimize for c. We'll sum up the squares of the grey lines and minimize those.

We can think of optimizing **c** as making the grey lines rubber bands, that pull on the line representing **c** (which pivots around the origin).

This is a lot like linear regression, but the task is slightly different (note that there is not target attribute here).

objective

$$
\begin{aligned}
&\arg\min_{\mathbf{c}} \sum_{\mathbf{x}} \|\mathbf{x}' - \mathbf{x}\|^2 \\
&= \arg\min_{\mathbf{c}} \sum_{\mathbf{x}} \|z \cdot \mathbf{c} - \mathbf{x}\|^2 \\
&= \arg\min_{\mathbf{c}} \sum_{\mathbf{x}} \|\mathbf{x}^\top \mathbf{c} \cdot \mathbf{c} - \mathbf{x}\|^2 \\
&= \arg\min_{\mathbf{c}} \sum_{\mathbf{x}} \sqrt{\sum_{i} \left(\mathbf{x}^\top \mathbf{c} \cdot c_i - x_i\right)^2}^2 \\
&= \arg\min_{\mathbf{c}} \sum_{\mathbf{x},i} \left(\mathbf{x}^\top \mathbf{c} \cdot c_i - x_i\right)^2
\end{aligned}
$$

such that $\|\mathbf{c}\| = 1$

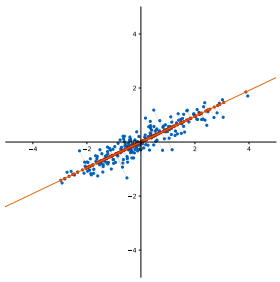
82

To find **c**, we will simply state our goals as an optimization objective. We want to find the **c** for which the squared distance between the data and the reconstructed data is minimized. We first fill in the definition of the reconstruction, and then the definition of the optimal z.

In the definition of the Euclidean distance, the square root cancels our against the square in ouroptimization, so that we are left with a sum of the squares over every dimension i in every reconstricted instance **x'**.

This leaves us with a simple objective to which we can apply any search algorithm, like gradient descent. One thing we must remember: we assumed that **c** is a unit vector.

This means we have an optimization problem with *a constraint*. This is a technical subject, that we'll see more of in lecture 6. For now, we can solve this problem by applying gradient descent and normalizing the vector **c** after every gradient update. This is called the projection method for constrained optimization. It doesn't always work, but it does here.



the first *principal* component $\mathbf{c}_1$

83

We run gradient descent and this is the solution that we find. It looks pretty good. It's hard to imagine any other line c leading to shorter grey lines.

We call **c** the **first principal component** of the data.

---

for more dimensions

repeat the process on the remaining directions.

the second principal component $\mathbf{c}_2$ is the unit vector

- orthogonal to $\mathbf{c}_1$
- which, together with $\mathbf{c}_2$, minimises the loss

84

If we want to reduce the dimensionality to more than one dimension, we repeat the process. Keeping the first principal component fixed, the second principal component is the one orthogonal to the first that minimizes the reconstruction loss given two reduced dimensions.

Each next principal component is the direction orthogonal to the previous ones, that minimizes the loss, when the data is reconstructed using all of them.

another perspective

The first principal component is the direction in which the *variance* is maximal.

The second principal component is the direction in which the remainder of the variance is maximal.

And so on.

85

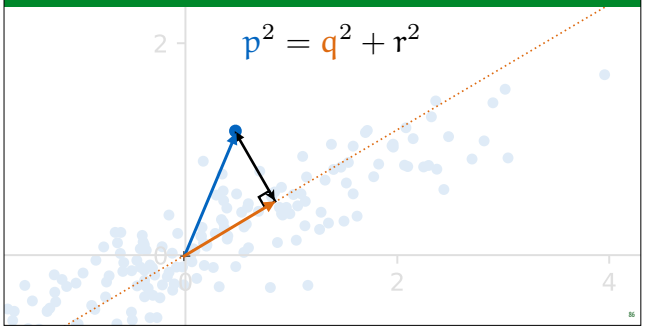
If you've heard about PCA before, you may be surprised by this definition using reconstruction loss. Usually, the principal components are defined as the directions in which the *variance* of the projected data is maximized.

The first principal component is the line along which the variance of the data is maximal when project onto the line. The second principal component is the line orthogonal to the first for which the variance is maximal, and so on.

It turns out, these two definitions are equivalent.

---

least squares *is variance maximisation*



86

Let's look at the one dimensional reduction again.

The variance of a one-dimensional dataset is defined as the average of the squares of all the distances to the data-mean. In our case, both the data and the reduction are mean-centered, so the variance is just the sum of all the squares of the z's; our reduced representations. In this picture, the length of the orange vector.

Thus, maximising the variance, means choosing c so that the (squared) length of the orange vector is maximized

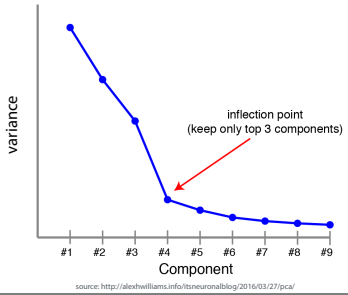
This arrangement into a right-angled triangle means that the magnitude of the original data (p, the squared distance to the mean) is related to the variance of the projected data (q) and the reconstruction error (r, in black) by the Pythagorean theorem.

Since p, the magnitude of the original data, is a constant, $q^2 + r^2$ is constant, and minimizing the squared reconsturction error r^2 is equivalent to maximising the variance of the projected data q^2.

In this setting we often talk about how much variance the reduced data *retains*, seeing the variance as a kind of "information content" in a representation of the data. A perfect reconstruction has the same total variance as the data.

---

choosing the nr. of principal components



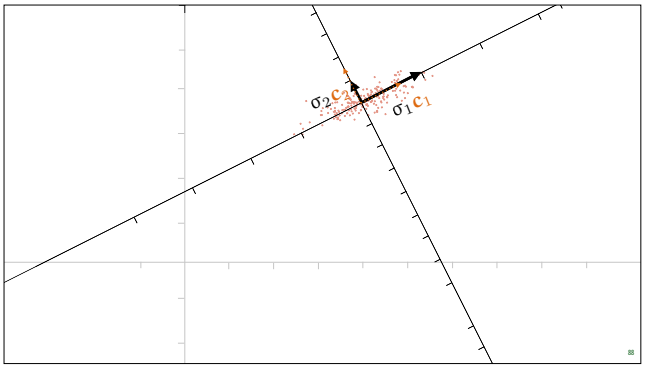
source: http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/

87

To apply PCA, we need to choose the number of dimensions to reduce to. We can just treat this as a hyperparameter and test different values.

But if we plot the variance or the reconstruction loss against the number of components, we often see a natural *inflection* point. In this case, we can retain the majority of the variance in the data by keeping only the first three principal components. The higher components add a little variance each, but not much.

What happens if we keep going until the new data has the same number of features as the original?



If we do that, we get perfect reconstructions, but our z's are still different from the original coordinates. We end up expressing the dat in another **basis**. It turns out, that this actually gives us a **whitening** of the data: in the new basis, the data is uncorrelated, with variance 1 along each axis.

---

yet another perspective

PCA *whitening*:

- apply PCA with k=m
- Use $\sigma_i \mathbf{c}_i$ as basis vectors

89

This way of whitening is called **PCA whitening**. We apply PCA with the same number of target dimensions as data dimensions. This gives us an orthonormal basis in which the data is uncorrelated. If we then measure the standard deviation along each component and multiply the basis vectors by that, we get a basis in which the data is whitened.

Note that while σ and $\mathbf{c}$ together is not an orthonormal basis, $\mathbf{c}$ by itself is. Thus, we can still easily transform back and forth between the whitened basis and the original data coordinates.

---

but wait…

Doesn't PCA have something to do with eigenvectors?
Or singular value decompositions?

Only if you want to compute it *efficiently*.
Or understand it even better. See peterbloem.nl/blog/pca

90

If you've heard about PCA before, you may be wondering why I haven't discussed eigenvector, or singular value decompositions. These topics are only necessary if you want to know the deeper workings of PCA, and if you want to compute it efficiently.

Computing PCA by gradient descent, one component at a time is illustrative, but in practice, there are far more efficient and precise ways to do it.

PCA

Principal Component Analysis

- linear transformation that minimises reconstruction loss
- linear transformation that maximises variance
- whitening transformation
- orders dimensions by variance captured/rec loss
  The first dimension is the most important for reconstructing the data, then the second and so on.

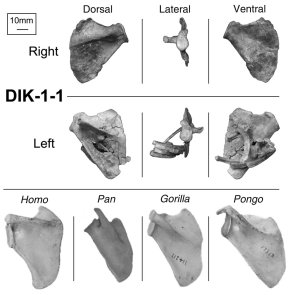
91

So what's the point?

92

This may seem like a lot of math for something so simple as reducing the dimensionality of a dataset.

But it turns out that these principal components are actually extremely versatile, and can give us a lot of insight into our data.



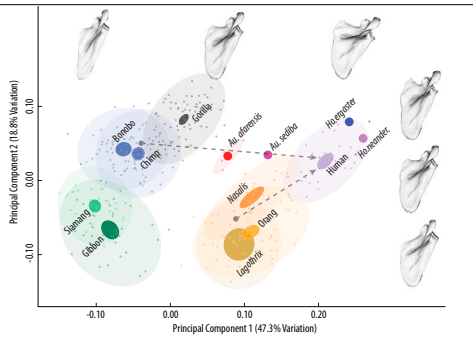
source: Green, David J., and Zeresenay Alemseged. **Australopithecus afarensis scapular ontogeny, function, and the role of climbing in human evolution.** *Science* 338.6106 (2012): 514-517.

93

We'll start with an example of how PCA is often used in research. Imagine you're a palaeontologist, and you find a shoulder bone, belonging to some great ape.

If you are a trained anatomist specialising in primates, you can easily tell for a single shoulder bone whether it's an early hominin fossil, which is a very rare find, or a chimpanzee fossil which isn't rare. But how do you then substantiate this? "It's true because I can see that it is" is not very scientific.

image source: **https://science.sciencemag.org/content/338/6106/514.full**



source: Fossil hominin shoulders support an African ape-like last common ancestor of humans and chimpanzees. Nathan M. Young, Terence D. Capellini, Neil T. Roach and Zeresenay Alemseged *http://www.pnas.org/content/112/38/11829*

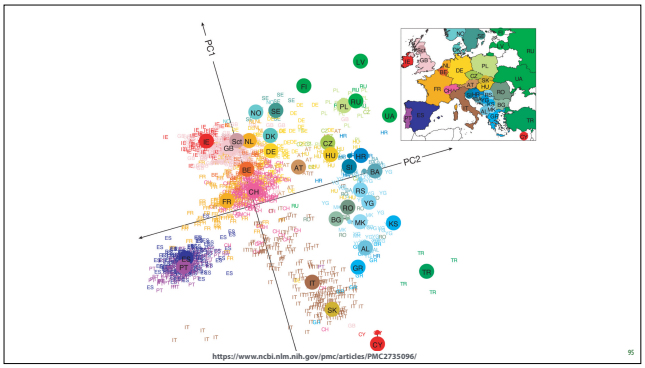
94

Here's one common approach. Take a large collection of the same specific bone (the *scapula,* or shoulder blade, in this case) from different apes and humans, and take a bunch of measurements (features) of each. Do a PCA, and plot the first two principal components. As you can see, the different species form very clear clusters, even in just two dimensions.

When we find a new fossil, we can see where it ends up in this space, and we can then show that what we've found is clearly closer to human than to chimp just by measuring it, and projecting it into this space.

Note also, that this data gives us some clues about how humans might have developed. The proto-humans Australopithecus Afarensis and Australopithecis Sediba, are both on a straight line between the cluster of Bonobo's, Chimps and Gorillas and the point where modern humans. These are indeed the great apes considered to be most like the ones from which we developed.

source: Fossil hominin shoulders support an African ape-like last common ancestor of humans and chimpanzees. Nathan M. Young, Terence D. Capellini, Neil T. Roach and Zeresenay Alemseged **http://www.pnas.org/content/112/38/11829**



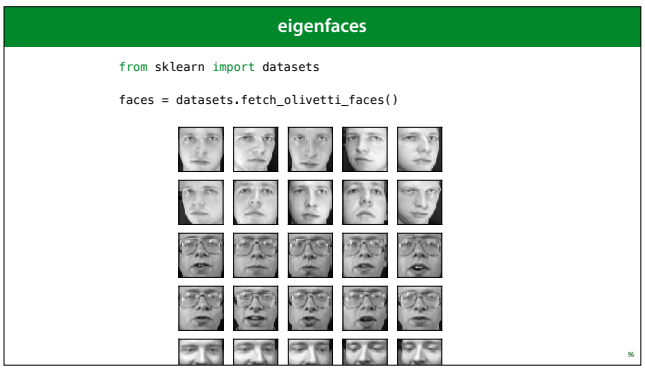
Here is another example of what PCA can tell us about a high-dimensional dataset.

In this research, the authors took a database of 1387 Europeans and extracted features from their DNA. They used about half a million sites on the DNA sequence where DNA varies among humans (i.e. 1387 instances: people, and 500k features: DNA markers). They also recorded where their subjects (or their immediate ancestors) were from.

Only the DNA data was fed to the PCA algorithm, with the person's origin only used afterward to color the points.

It turns out that the two principal components of this data largely express how far north the person lives, and how far east the person lives, which means that if you plot the data in the first two principle components, you get a fuzzy picture of Europe.

In short, the large scale geography of Europe can be extracted from our DNA. If I sent a large sample of European DNA to some aliens on the other side of the galaxy who'd never seen our planet, they could use it to get a rough idea of our geography.

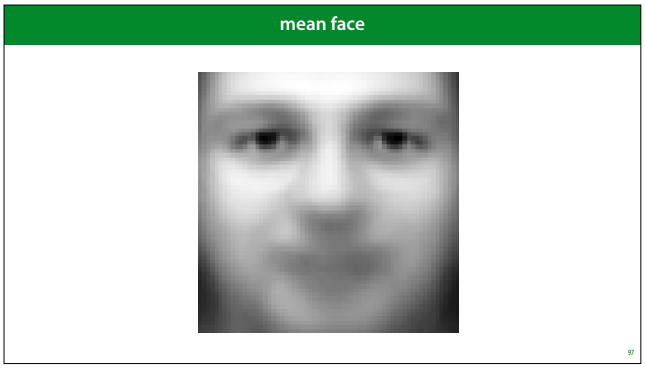


Finally, possibly the most magical illustration of PCA: **eigenfaces**.

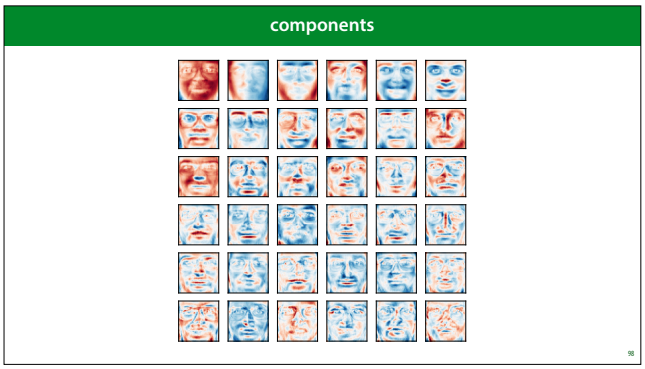
Here we have a dataset (which you can easily get from sklearn) containing 400 images, in 64x64 grayscale, of a number of people. The lighting is nicely uniform and the facial features are always in approximately the same place.

We take each pixel as a feature, giving us 400 instances each represented by a 4096-dimensional feature vector.

The prefix eigen- comes from the eigendecomposition often used to derive the PCA analysis. It's out of scope for us, but you can read more about this here: **peterbloem.nl/blog/pca-2**.

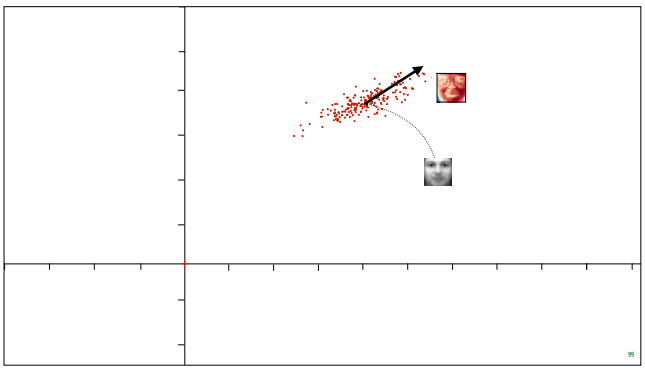


Here is the sample mean of our data, re-arranged back into an image.



Once we have the principal components, each 4096-dimensional vector, we can take their values, assign them a color, like red for negative values, blue for positive values, and re-arrange them back into images. Remember, every dimension represents a pixel.

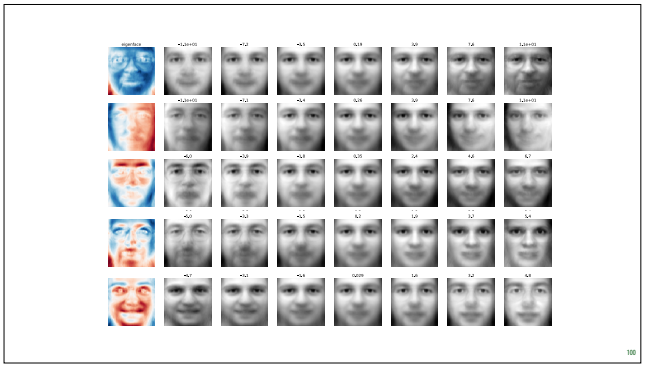
These are the first 30 principal components displayed this way (top left is the first, to the right of that is the second and so on).



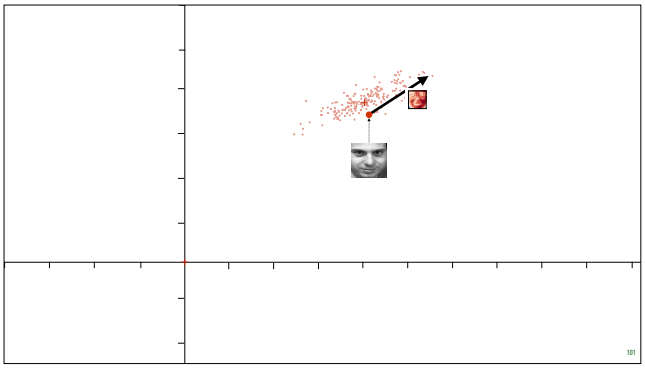
This is one way to interpret the principal components. They are the basis vectors that are most natural for our data. Remember, PCA is also a whitening operation.

The first principal component is the direction that captures most of the variance of our data. Or, projecting our data down to the first principal component gives us the lowest reconstruction error.

We can visualize this space, by starting at the data mean, and adding a small bit of the nth principal component.

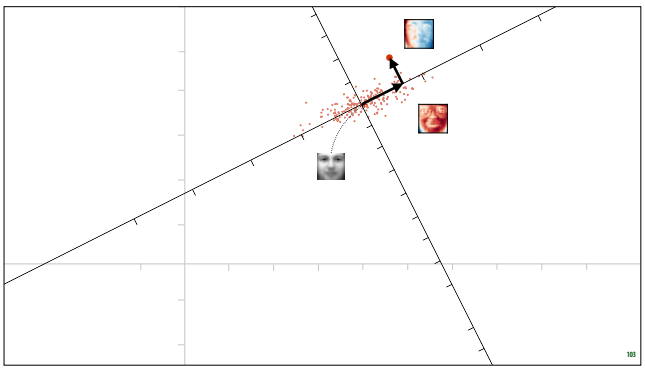


Starting from the mean face (in the middle column), we take little steps along the direction of one of our principal components (or in the opposite direction). We see that moving along the first principal component roughly corresponds to ageing the face. Moving along the fourth seems to make the face more feminine.



Because the reconstruction is a linear projection (details in the reading materials) we can also add the principal components to instances in our data directly.

The middle column represents the starting point. To the right we add the k-th principal component, to the left we subtract it. Note, in particular the effect of the fifth principal component: subtracting it opens the mouth, and adding it seems to push the lips closer together.



To reconstruct a point, we start with the mean, and add a bit of the first principal component, then of the second principal component and so on.

If we think of our principal components as a new **basis** for our data, then we are just looking up our point by first moving some distance along the first axis, then along the second axis and so on. Just like we would look up a point given its coordinates in the standard basis.

Here's what that looks like. top left is the mean. To the right is the reconstruction from just the first principal component. Next is what we get is we add the second principal component to that and so on.

105

After 60 principal components, the image starts to look pretty recognizable. We've reduced our data from 4096 dimensions to just 60 dimensions, and still retained enough information to tell people apart.

---

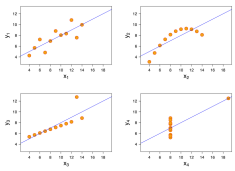
recap

PCA: dimensionality reduction

Using linear transformations
We'll discuss nonlinear versions in lecture 9

Principal components are a *natural basis* for your data
Good way of getting insight into your data

106



mlcourse@peterbloem.nl

And getting insight into your data, as we said at the start of the lecture, is what it's all about.